Swift: Fast, Reliable, Loosely Coupled Parallel Computation

Yong Zhao,[1] Mihael Hategan,[2] Ben Clifford,[2] Ian Foster,[1,2,3]
Gregor von Laszewski,[2,3] Ioan Raicu,[1] Tiberiu Stef-Praun,[2] Mike Wilde[2,3]
[1] *Department of Computer Science, University of Chicago, Chicago, IL 60637, USA*
[2] *Computation Institute, University of Chicago & Argonne National Laboratory*
[3] *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA*
{yongzh,iraicu}@cs.uchicago.edu, benc@hawaga.org.uk, {foster,gregor,hategan,tiberius,wilde}@mcs.anl.gov

Abstract

We present Swift, a system that combines a novel scripting language called SwiftScript with a powerful runtime system based on CoG Karajan, Falkon, and Globus to allow for the concise specification, and reliable and efficient execution, of large loosely coupled computations. Swift adopts and adapts ideas first explored in the GriPhyN virtual data system, improving on that system in many regards. We describe the SwiftScript language and its use of XDTM to describe the logical structure of complex file system structures. We also present the Swift runtime system and its use of CoG Karajan, Falkon, and Globus services to dispatch and manage the execution of many tasks in parallel and Grid environments. We describe application experiences and performance experiments that quantify the cost of Swift operations.

1. Introduction

A common pattern in scientific computing involves the execution of many tasks that are coupled only in the sense that the output of one may be passed as input to one or more others—for example, as a file, or via a Web Services invocation. While such "loosely coupled" computations can involve large amounts of computation and communication, the concerns of the programmer tend to be different than in traditional high performance computing, being focused on management issues relating to the large numbers of datasets and tasks (and often, the complexities inherent in "messy" data organizations) rather than the optimization of interprocessor communication.

Consider this painful but familiar scenario: *A neuroscientist needs to analyze ten thousand functional magnetic resonance imaging (fMRI) files. The analysis program is a complex Perl script. Files are stored in a collection of UNIX directories, with metadata coded in directory and file names. Local computing facilities are inadequate. Thus, the scientist must manually extract files, copy them to a remote cluster, start a home-grown script to dispatch tasks, and check exit codes and output files to see which tasks succeeded and failed. And when the computation is completed, the problem remains of documenting what was done*.

Such difficulties motivated our design of Swift, a parallel programming system that integrates the following elements to address these difficulties:

- A scripting language, **SwiftScript**, allows users to express operations on datasets in terms of their logical organization; the XML Dataset Typing and Mapping (**XDTM**) [10] notation is used to define a mapping between that logical organization and the underlying physical structure.
- An execution engine, CoG **Karajan** [9], a lightweight provisioning and submission system, **Falkon** [13], and a compiler and associated libraries, execute tasks specified via SwiftScript programs on local or remote computers.
- A provenance-recording component, **Kickstart** [18], captures execution details for diagnosis and eventual recording in a provenance database.

These elements allow a few lines of SwiftScript to specify computations involving large numbers (tens or hundreds of thousands) of files and tasks, and for those computations to be executed efficiently and reliably on many distributed computers. The impact on end users such as our unfortunate neuroscientist can be enormous. Code sizes can be reduced by an order of magnitude or more [19]. In one example, a 160-member climate model ensemble took 2.5 months when performed manually; a 250-member ensemble was finished within 4 days—admittedly on a faster computer—when automated with a precursor to Swift [11]. Other users are found in the physical, biological, and social sciences, and in the humanities and science education.

Swift grew out of the Virtual Data System (VDS) [7], which integrated a simple virtual data language, planners (including Pegasus [6]) for program

optimization and scheduling, DAGMan for task management [4], kickstart, and a virtual data catalog [20]. Swift improves on VDS in its use of XDTM to define logical views of datasets; SwiftScript and CoG Karajan support for iteration operations, which allow for more concise specifications of computations over larger datasets; and Falkon for efficient task dispatch.

The rest of this paper is organized as follows. We introduce SwiftScript and the Swift system design in Sections 2 and 3. We present system evaluation results and applications in Sections 4 and 5. We discuss related work in Section 6, and summarize in Section 7.

2. Notation: SwiftScript and XDTM

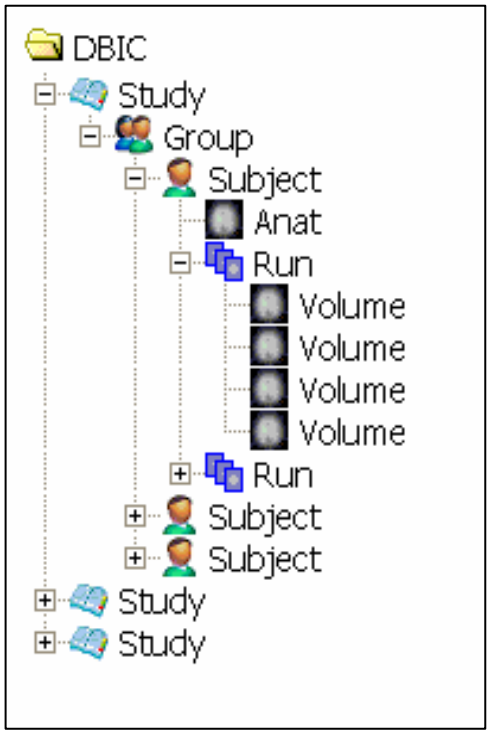
The need to process numerous tasks reliably and efficiently arises, for example, when performing large-scale data analysis or executing many computations to study sensitivity to parameter values (in parameter studies) and/or initial conditions (in ensemble simulations). Users often struggle with bookkeeping tasks due to numerous tasks, datasets, and resources.

Users can benefit from a concise and readable notation that simplifies the description, maintenance, and debugging of problem specifications. Such a notation can also facilitate high-performance execution by revealing opportunities for concurrent execution. Conventional scripting languages such as UNIX Shell or Perl, frequently used to implement the applications that we target, are not concise, readable, easily parallelizable or analyzable, and are not amenable to the automation of provenance tracking. We overcome these problems with XDTM and SwiftScript.

2.1. XDTM

Even logically simple applications can become complicated when they "messy" data is stored in odd formats and storage organizations. For example, compare the logical and physical layouts in Figure 1. The *logical* organization is a clean hierarchy of studies, groups, subjects (patients), runs (series of volumes), and volumes (brain scans), while the *physical* layout is a complex mix of directory structures and files [8]. (It is not obvious that '/knottastic' is a *Study*, containing *Group* 'AA', in turn containing *Subject* '04nov06aa,' etc.) The result, without Swift, is often complex and hard-to-maintain application orchestration code.

We address this problem by using XDTM, which allows logical datasets to be defined in a manner that is independent of the datasets' concrete physical representations. XDTM employs a two-level description of datasets, defining separately via a type system based on XML Schema the abstract structure of datasets, and the mapping of that abstract data structure to physical representations.




```
./knottastic
total 58
drwxr-xr-x  4 yongzh users 2048 Nov 12 14:15 AA
drwxr-xr-x  4 yongzh users 2048 Nov 11 21:13 CH
drwxr-xr-x  4 yongzh users 2048 Nov 11 16:32 EC

./knottastic/AA:
total 4
drwxr-xr-x  5 yongzh users 2048 Nov  5 12:41 04nov06aa
drwxr-xr-x  4 yongzh users 2048 Dec  6 12:24 11nov06aa

. /knottastic//AA/04nov06aa:
total 54
drwxr-xr-x  2 yongzh users  2048 Nov  5 12:52 ANATOMY
drwxr-xr-x  2 yongzh users 49152 Dec  5 11:40 FUNCTIONAL

. /knottastic/AA/04nov06aa/ANATOMY:
total 58500
-rw-r--r--  1 yongzh users      348 Nov  5 12:29 coplanar.hdr
-rw-r--r--  1 yongzh users 16777216 Nov  5 12:29 coplanar.img

. /knottastic/AA/04nov06aa/FUNCTIONAL:
total 196739
-rw-r--r--  1 yongzh users     348 Nov  5 12:32 bold1_0001.hdr
-rw-r--r--  1 yongzh users  409600 Nov  5 12:32 bold1_0001.img
-rw-r--r--  1 yongzh users     348 Nov  5 12:32 bold1_0002.hdr
-rw-r--r--  1 yongzh users  409600 Nov  5 12:32 bold1_0002.img
-rw-r--r--  1 yongzh users     348 Nov  5 12:32 bold1_0003.hdr
-rw-r--r--  1 yongzh users  409600 Nov  5 12:32 bold1_0003.img
```

Figure 1: fMRI logical data structure (left) vs. physical file system layout (right)

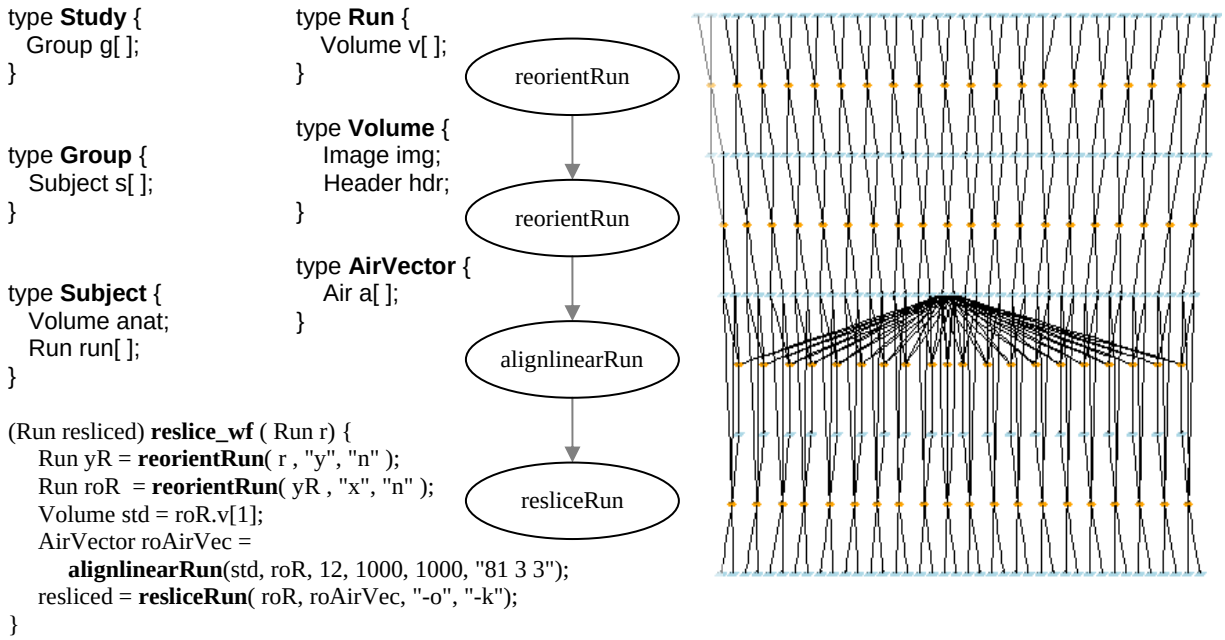


Figure 2 A Swift program (fragment) and the resulting task graph

A dataset's *logical structure* is specified via a subset of XML Schema, which defines primitive scalar data types such as Boolean, Integer, String, Float, and Date, and also allows for the definition of complex types via the composition of simple and complex types. The use of XML Schema as a type system has the benefit of supporting powerful standardized query languages such as XPath in our selection methods.

A dataset's *physical representation* is then defined by a mapping descriptor, which describes how each element in the dataset's logical schema is stored in/fetched from physical structures such as directories, files, and database tables. To permit reuse for different datasets, mapping descriptors may refer to external parameters for such things as dataset location(s).

We use a virtual integration approach to implement the mapping mechanism. Each data source is regarded as a virtual XML source, with its structure described in an XML Schema. A mapper is responsible for accessing the data source and converting its data to/from an XML document or stream that conforms to the XML schema. The case is somewhat different from a traditional data integration approach, since we need to deal with writing/updating to data sources as well as querying them.

We define a standard mapping interface so that different data providers can implement the interface to support access to various data representations. We provide default mapping implementations for string mapping, file system mapping, and CSV (comma separated-value) files.

2.2. SwiftScript

The SwiftScript scripting language builds on XDTM to allow for the definition of typed data structures and procedures that operate on such data structures. SwiftScript procedures define logical data types and operations on those logical types; the SwiftScript implementation uses mappers to access the corresponding physical data. In addition to providing the usual advantages of strong typing (type checking, self-documenting code, etc.), this approach allows SwiftScript programs to express opportunities for parallel execution easily, for example by applying transformations to each component of a hierarchically defined logical structure.

As an example, the logical structure of the fMRI dataset shown in Figure 1 can be represented by the SwiftScript type declarations in the upper left of Figure 2. Here, *Study* is declared as containing an array of *Group*, which in turn contains an array of *Subject*, etc. Similarly, an fMRI *Run* is a series of brain scans called volumes, with a *Volume* containing a 3D image of a volumetric slice of a brain image, represented by an *Image* (voxels) and a *Header* (scanner metadata).

Figure 3 includes two example procedures. We examine *reorientRun* first. This is what we call a compound procedure, meaning it calls one or more other SwiftScript procedures. Note the typed input arguments (to the right of the procedure name) and output argument (to the left). The procedure takes in a run *ir* and applies the procedure *reorient* (which rotates a brain image along a certain axis) to each volume in

the run to produces a reoriented run *or*. Because the multiple calls to *reorient* operate on independent data elements, they can proceed in parallel.

The procedure *reorient* in Figure 3 is *atomic*, corresponding to an invocation of an executable program or a Web Service. This procedure has typed input parameters *iv*, *direction* and *overwrite* and one output *ov*. The body of this particular procedure specifies that it invokes a program (conveniently, also called reorient) that will be dynamically mapped to a binary executable. (This executable will execute at an execution site chosen by the Swift runtime system.) The body also specifies how input parameters map to command line arguments. The notation *@filename* is a built-in mapping function that maps a logical data structure to a physical file name.

```
(Volume ov) reorient (Volume iv, string direction,
                                    string overwrite) {
    app {
        reorient  @filename(iv.hdr)
                  @filename(ov.hdr)
                  direction
                  overwrite;
    }
}

(Run or) reorientRun (Run ir, string direction,
                                    string overwrite) {
    foreach Volume iv, i in ir.v {
        or.v[i] = reorient (iv, direction, overwrite);
    }
}
```

Figure 3 fMRI procedure declarations

A compound procedure can also comprise a series of procedure calls, using variables or datasets to establish data dependencies. Such procedures can themselves be called by other procedures, thus defining a potentially large and complex execution graph.

The procedure *reslice_wf* (Figure 2, lower left) applies *reorientRun* to a run first in the *x* axis and then in the *y* axis, and then aligns each image in the resulting run with the first image. The program *alignlinear* determines how to spatially adjust an image to match a reference image, and produces an *air* parameter file. The actual alignment is done by the program *reslice*. Note that variable *yR*, being the output of the first step and the input of the second step, defines the data dependencies between the two steps. More complex procedures can be composed in a similar fashion, using iterations and other constructs.

The *reslice_wf* example defines a simple four-step pipeline computation. The pipeline is illustrated in the center of Figure 2, while on the right we show the expanded graph for a 20-volume run. Each volume comprises an image file and a header file, so there are a total of 40 input files and 40 output files. We can also apply the same procedure to a run containing hundreds or thousands of volumes.

SwiftScript allows concise definitions of logical data structures and logical procedures that operate on them, and complex computations to be composed from simple and compound procedures. Its support for nested iterations can allow a compact SwiftScript program (for example, a nested set of iterations that applies the program *reorient* to each volume in a whole *Study*) to express hundreds of thousands of parallel tasks. We have shown that SwiftScript programs can be at least an order of magnitude smaller in lines of code than other approaches such as Shell scripts and directed acyclic graph specifications [19].

3. Implementation

The Swift runtime system (see Figure 4) is a scalable environment for efficient specification, scheduling, monitoring and tracking of SwiftScript programs. We describe its components one by one.

Program specification: computations defined in SwiftScript programs are compiled by a SwiftScript compiler into abstract computation plans, which can be scheduled for execution by the execution engine.

Scheduling: Swift uses CoG Karajan as its execution engine. Karajan provides libraries for data transfer, task submission, and Grid services access. Such operations can be organized using language constructs such as sequential and parallel execution, sequential and parallel iterations, conditional execution and functional abstraction etc. We extend Karajan with libraries to support the XDTM type system and logical dataset manipulation, adapters to access legacy VDS components (for instance, site catalog), mappers for accessing heterogeneous physical data storage, and fault tolerance mechanisms. Karajan uses light-weight threading techniques to instantiate and dispatch tasks, and thus can execute large task graphs (see Section 4).

Execution: Abstract execution plans are interpreted and dispatched by Karajan onto execution sites. These abstract plans do not specify execution location: tasks are dispatched to virtual nodes, which can be bound variously to personal desktops, clusters, and distributed Grids. Specifics such as site selection, data stage-ins and stage-outs, and error checking are determined at runtime. Callouts allow customized functions to determine where to dispatch tasks, how to group tasks to increase granularity, and/or when and how to perform data staging operations. Swift also supports advanced scheduling and optimization methods, such as load balance, fault tolerance, and computation restart.

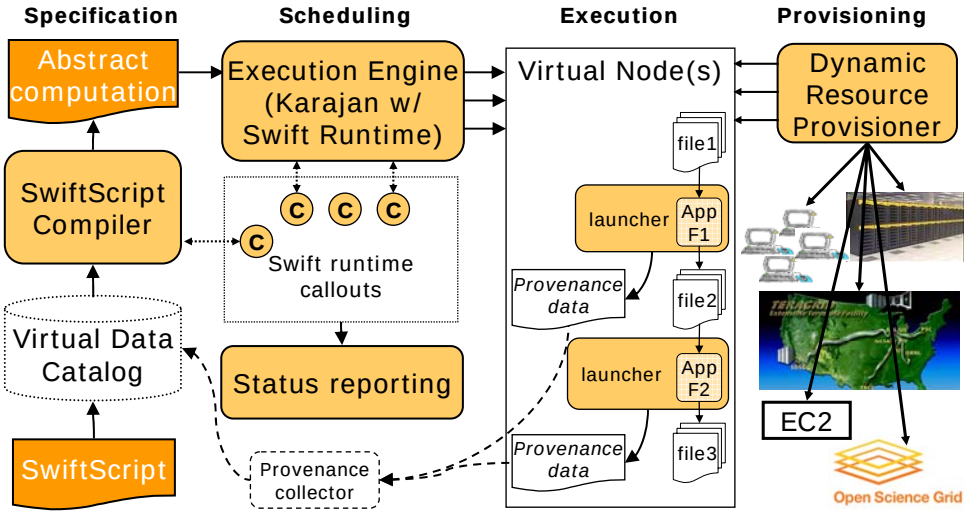


Figure 4 Swift system diagram

Provenance tracking: Individual tasks are invoked by a launcher program (e.g., kickstart) which monitors execution and gathers provenance information. We plan to record this information in a virtual data catalog, as in VDS.

Provisioning: Tasks determined to be executable by Karajan can be submitted directly, via a GRAM submission, to a (local or remote) scheduler for execution; in this way, resource provisioning and task submission are handled together. The Swift architecture also allows for those two functions to be separated. Specifically, a dynamic resource provisioner (right hand side of figure) can interact with local or remote computing systems to get computing resources, and then deploy task execution services onto those resources. The execution services then interact with a task queue service to obtain tasks to execute. Falkon provides implementations of these functions.

4. Evaluation

We report on experiments that evaluate the performance of various elements of the Swift system.

We consider first the size of the computations that can be executed correctly. Figure 5 shows the maximum number of tasks (measured by the number of nodes in a task graph) that Swift can process and dispatch with certain amount of available memory. The system can support about 4000 nodes with 32MB of memory and 160,000 nodes with 1GB memory.

Swift addresses reliability issues at several levels. At the software development level, its type checking capabilities allow it to identify potential problems in a program prior to execution. Its support for virtual nodes makes it easy to first test a program on a local host with a small set of datasets, and then move to larger problems and execution sites.

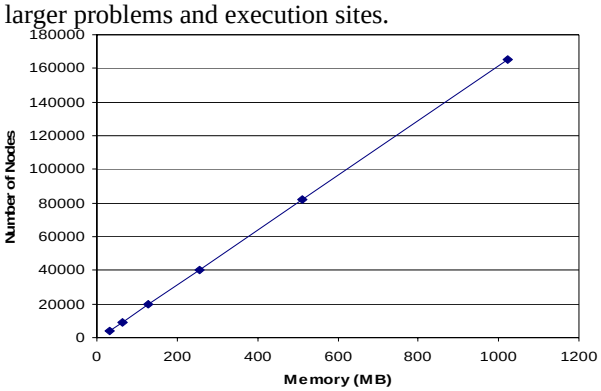


Figure 5 Swift system scalability

During execution, the underlying Karajan engine supports flexible exception handling mechanism. Transitory problems are recovered by retrying the faulty tasks (for instance, retry a transfer if a GridFTP server is busy), and host level faults (where a resource exhibits problems with unknown duration) are dealt with by rescheduling a task on a different site.

Swift also keeps a restart log, allowing it to resume the state of a computation in case of premature termination (for instance, caused by a machine reboot). We have tested restartability by repeatedly interrupting program execution, and verified that our programs continue from where they were interrupted. We also note some (good) side effects to this mechanism: (1) new inputs can be added after a computation has been run for some time, and once we restart the computation, the system is able to figure out that these new inputs are present and not processed, and thus schedule their executions. (2) We can make modifications to a program and restart it, as long as the modifications do not affect data flows that have already happened. This

effect is useful for debugging and testing purposes. The Swift restart log is similar to a Condor rescue DAG, except that Condor tags tasks that are finished, whereas we log datasets that are produced successfully.

Swift uses three mechanisms to improve efficiency:

Pipelining: the immediate dispatch of dependent tasks, even if in a different "foreach" block. Swift's data-driven model means that once an item in a collection is processed, any processes dependent on that item can proceed immediately, without waiting for the whole collection to finish.

Clustering: many scientific computations comprise many short tasks. (For example, the reorient program in the fMRI example runs in a few seconds.) The initialization and scheduling of such tasks can pose significant overheads. Thus, we (optionally) bundle groups of (mostly independent) tasks and submit them as a single task.

Pluggable execution providers: The CoG Karajan resource provider interface allows Swift to schedule tasks to computers via different mechanisms. We used existing local host, cluster scheduler, and GRAM providers, and implemented a new provider for the Falkon (Fast and Lightweight Task Execution) tools [13]. Falkon supports the efficient execution of many small tasks in batch scheduled environments.

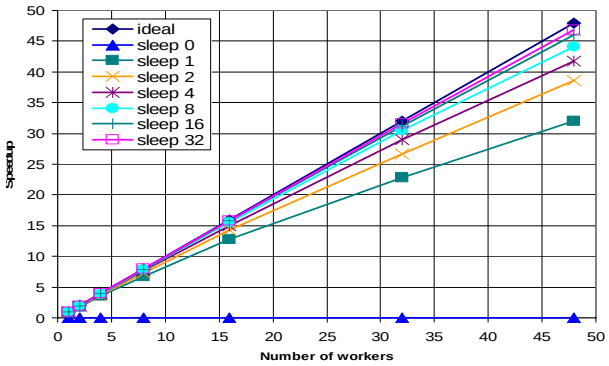


Figure 6 Swift speedup with Falkon provider

Figure 6 shows speedups measured for synthetic tasks with Falkon. Swift executed computations comprising 960 test ("sleep") tasks from a submit host at the University of Chicago (UC_SUB) to a Falkon service running on the TeraGrid ANL cluster. We varied the sleep durations and also configured Falkon to use different number of nodes for task execution. The ideal speedup is equal to the number of nodes used. We see that Falkon can operate efficiently even when tasks are short. On 48 nodes, Swift+Falkon can achieve close to 47 times speedup for 32-second tasks.

Figure 7 shows Swift throughput with the Falkon provider. Falkon ran on the TeraGrid ANL site and we submitted both from a separate node within that cluster (ANL→ANL) and from UC_SUB (UC→ANL). We measured throughput, defined as the number of sleep(0) tasks completed per second. Swift was able to achieve up to 56 tasks/second in the LAN setting and 46 tasks/second in the remote setting. The scheduling of each sleep task actually involved many extra steps such as site selection, execution directory set up, exit code checking, and clean up. However, we improve throughput relative to GRAM+ PBS (which achieves around two tasks per second) by a factor of 23.

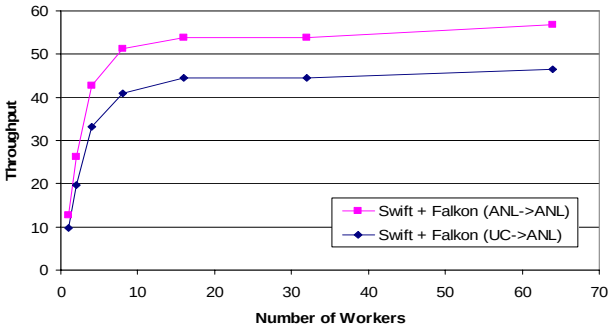


Figure 7 Swift throughput with Falkon provider

We also measured turnaround time for the fMRI workflow of Figure 1, with various input sizes (number of volumes) and different scheduling strategies. We submitted from UC_SUB to the TeraGrid ANL cluster. Results are in Figure 8; error bars indicate the standard deviation of measurements.

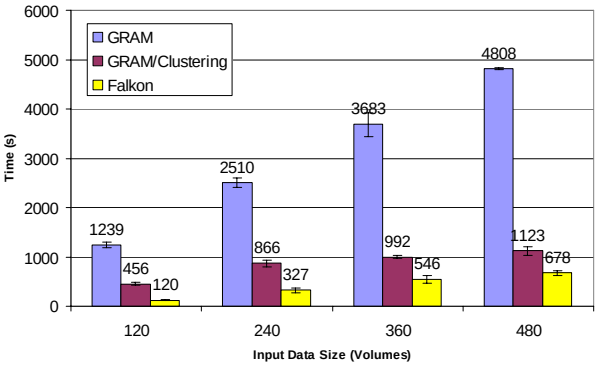


Figure 8 Execution time for the fMRI workflow

Since for each volume, each individual task required just a few seconds, it is inefficient to schedule each task over GRAM+PBS, since the overhead of PBS resource allocation is large relative to the short execution time. GRAM+PBS submission had low throughput although it could have potentially used all the available nodes on the site. With clustering, execution time was reduced by up to four times (tasks were bundled into roughly 8 groups), as the overhead was amortized by the bundled tasks. Falkon (with 8 worker nodes) further reduced execution time by 40-70%, as each task was dispatched efficiently.

5. Swift Applications

Swift has been applied to applications in the physical, biological, and social sciences, the humanities, and science education. We summarize some applications and their scales in Table 1. For each, we give the number of tasks in a typical analysis run and the number of levels (or stages) in such analyses.

6. Related Work

The MapReduce [5] tool for parallel computations is limited to processing key-value based data, and the runtime environment requires Google File System. Swift targets scientific applications that process heterogeneous data formats, and can schedule computations in a location-independent way.

Table 1: Example Swift applications

Application	#Tasks/ Run	#Levels
fMRI AIRSN Processing	100s	12
fMRI Aphasia Study	500	4
NVO/NASA Photorealistic Montage	1000s	16
QuarkNet/I2U2 Physics Science Education	10s	3-6
Radiology Classifier Training	1000s	5
SIDGrid EEG Wavelet Processing, Gaze Analysis	100s	20

Pegasus [6] and DAGMan [4] can also schedule large scale computations in Grid environments. DAGMan provides a workflow engine that manages Condor tasks organized as directed acyclic graphs (DAGs) in which each edge corresponds to an explicit task precedence. It has no knowledge of data flow, and in distributed environments works best with a higher-level, data-cognizant layer. DAGMan also lacks dynamic features such as iteration or conditional execution. Pegasus is primarily a set of DAG transformers that can translate a workflow graph into a location-specific DAGMan input file; prune tasks for files that exist; select sites for tasks; and cluster tasks based on various criteria. The planners must operate on an entire workflow statically, and execution sites may not be changed after a workflow is processed, which can be long before a task runs, a strategy that may not work well in dynamic environments.

BPEL [3] is used primarily for service composition and orchestration. Early versions lacked support for iteration which would result in large programs; this problem is addressed in the new version 2.0. In addition, its complex XML is cumbersome to write compared with our compact scripting language.

Taverna [12], Triana [17], and Kepler [1] have also been applied to scientific problems. However, they do not abstract dataset types or provide location transparency. Data movement and Grid task submission all need to be specified explicitly. Their support for multi-site Grid execution is limited.

As discussed earlier, Swift uses CoG Karajan [9] libraries and primitives for task scheduling, data transfer, and Grid task submission. Swift adds support for high-level abstract specification of large parallel computations, data abstraction, and workflow restart, and also (via Falkon and Globus) fast, reliable execution over multiple sites.

7. Summary and Future Work

Swift addresses important end-to-end issues in large-scale loosely coupled parallel computation. It imposes elegant order on a messy complex world of distributed and failure-prone applications. It provides a clean separation of logical data structures and physical storage formats, a scripting language for concise specification and composition of complex workflows, and a scalable runtime system that can manage and dispatch hundreds of thousands of tasks onto a variety of parallel and distributed computation facilities.

Swift provides a wide range of capabilities to support the formulation, execution and management of large compute- and data- intensive computations:

Scalability: Swift has demonstrated large-scale execution of large computations on both parallel computers and distributed systems.

Scripting: Scientists often seem to prefer scripting languages. SwiftScript meets this need with a simple, familiar, and expressive notation.

Dataset typing and iteration: Swift allows the declaration of logical data structures and typed procedures that iterate over such datasets. Nested iterations can easily scale to thousands or even millions of data objects and associated tasks.

Dataset mapping is critical for automating task execution with location independence. Systems need to know how to access dataset components and how to pack datasets and transport them to an execution site. Swift's dataset layout description model allows users to work at a clean abstract level.

Application service interoperability: Swift can integrate both service-oriented and command-oriented applications. While services are gaining adoption, most scientists still use non service-oriented applications. Swift provides a bridge between these two models.

Provenance and annotation: Swift integrates provenance and annotation with computation through a language that makes data flow explicit and trackable, and a catalog (soon to be integrated) that records data derivation activities.

This unique combination of features enables the automation of scientific data and workflow managements, and improves usability and productivity in scientific applications and data analyses.

We continue to work to improve Swift's usability, functionality, and scalability. In particular, we are working on streamlining SwiftScript's application interface, and on integrating VDS data provenance structures to represent programs, metadata, and runtime provenance [19], to support a wide range of provenance queries [20]. We are also benchmarking system components with large scale application runs.

Acknowledgments

We used computers at the U.Chicago Computation Institute, and at the Teragrid U.Chicago site. This work was supported by the National Science Foundation GriPhyN Project (ITR-800864) and iVDGL (PHY-122557); I2U2: Interactions in Understanding the Universe (PHY-0636265); the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy (DE-AC02-06CH11357); and the National Institutes of Health (NS37470, NS44393, DC008638-01). We thank Veronika Nefedova, Rob Jacob, Steven Small, Uri Hasson, Dan Katz, Maryellen Giger, Andrew Jamieson, and the SIDGrid project (Bennett Bertenthal and Sarah Kenney: NSF BCS 05-37849) for their collaboration on the applications described here.

References

[1] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludäscher, B. and Mock, S., Kepler: An Extensible System for Design and Execution of Scientific Workflows. in *16th Intl. Conference on Scientific and Statistical Database Management*, (2004).

[2] Annis, J., Zhao, Y., Voeckler, J., Wilde, M., Kent, S. and Foster, I., Applying Chimera Virtual Data Concepts to Cluster Finding in the Sloan Sky Survey. in *SC2002*, (Baltimore, MD, 2002).

[3] Business Process Execution Language for Web Services, Version 1.0, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, 2002.

[4] Condor DAGMan (Directed Acyclic Graph Manager), http://www.cs.wisc.edu/condor/dagman, 2007.

[5] Dean, J. and Ghemawat, S., *MapReduce: Simplified data processing on large clusters*. In OSDI, 2004.

[6] Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., Laity, A., Jacob, J.C. and Katz, D.S. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming, 13* (3). 219-237.

[7] Foster, I., Voeckler, J., Wilde, M. and Zhao, Y., Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. in *14th Intl. Conf. on Scientific and Statistical Database Management*, (Edinburgh, Scotland, 2002).

[8] Horn, J.V., Dobson, J., Woodward, J., Wilde, M., Zhao, Y., Voeckler, J. and Foster, I. Grid-Based Computing and the Future of Neuroscience Computation. in *Methods in Mind*, MIT Press, 2006.

[9] Laszewski, G.v., Hategan, M. and Kodeboyina, D. Java CoG Kit Workflow. in Taylor, I.J., Deelman, E., Gannon, D.B. and Shields, M. eds. *Workflows for eScience*, 2007, 340-356.

[10] Moreau, L., Zhao, Y., Foster, I., Voeckler, J. and Wilde, M., XDTM: XML Data Type and Mapping for Specifying Datasets. in *European Grid Conference*, (2005).

[11] Nefedova, V., Jacob, R., Foster, I., Liu, Y., Liu, Z., Deelman, E., Mehta, G. and Vahi, K., Automating Climate Science: Large Ensemble Simulations on the TeraGrid with the GriPhyN Virtual Data System. in *2nd IEEE International Conference on eScience and Grid Computing*, (2006).

[12] Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M.R., Wipat, A. and Li, P. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows *Bioinformatics Journal*, *20* (17). 3045-3054.

[13] Raicu, I., Zhao, Y., Dumitrescu, C., Foster, I., Wilde, M., Falkon: a Fast and Light-weight tasK executiON framework, Argonne National Laboratory, Mathematics and Computer Science Division Preprint ANL/MCS-P1424-0507, May 2007..

[14] Sulakhe, D., Rodriguez, A., Wilde, M., Foster, I. and Maltsev, N., Using Multiple Grid Resources for Bioinformatics Applications in GADU. in *IEEE/ACM International Symposium on Cluster Computing and Grid*, (2006).

[15] Swift, http://www.ci.uchicago.edu/swift, 2007

[16] TeraGrid, http://www.teragrid.org, 2007.

[17] Taylor, I., Shields, M., Wang, I. and Harrison, A. Visual Grid Workflow in Triana. *Journal of Grid Computing,, 3* (3-4). 153-169.

[18] Vöckler, J.-S., Mehta, G., Zhao, Y., Deelman, E. and Wilde, M., Kickstarting Remote Applications. in *2nd International Workshop on Grid Computing Environments* (2006).

[19] Zhao, Y., Dobson, J., Foster, I., Moreau, L. and Wilde, M. A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data. *SIGMOD Record 34* (3). 37-43.

[20] Zhao, Y., Wilde, M., Foster, I., Applying the Virtual Data Provenance Model, Proceedings of the International Provenance and Annotation Workshop

2006 (IPAW2006), Lecture Notes in Computer Science, Springer, 2006.